NASA Contractor Report 187582
ICASE Report No. 91-45

ICASE

PERFORMANCE AND FAULT-TOLERANCE OF NEURAL NETWORKS FOR OPTIMIZATION

Peter W. Protzel
Daniel L. Palumbo
Michael K. Arras

Contract No. NAS1-18605
June 1991

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Performance and Fault-Tolerance of Neural Networks for Optimization*

Peter W. Protzel

Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665
Electronic-Mail Address: protzel@icase.edu

Daniel L. Palumbo

System Validation Methods Branch
Mail Stop 130, NASA Langley Research Center, Hampton, VA 23665

Michael K. Arras

Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665

Abstract

One of the key benefits of future hardware implementations of certain Artificial Neural Networks (ANNs) is their apparently "built-in" fault-tolerance, which makes them potential candidates for critical tasks with high reliability requirements. This paper investigates the fault-tolerance characteristics of time-continuous, recurrent ANNs that can be used to solve optimization problems. The performance of these networks is first illustrated by using well-known model problems like the Traveling Salesman Problem and the Assignment Problem. The ANNs are then subjected to up to 13 simultaneous "stuck-at-1" or "stuck-at-0" faults for network sizes of up to 900 "neurons." The effect of these faults on the performance is demonstrated and the cause for the observed fault-tolerance is discussed. An application is presented in which a network performs a critical task for a real-time distributed processing system by generating new task allocations during the reconfiguration of the system. The performance degradation of the ANN under the presence of faults is investigated by large-scale simulations and the potential benefits of delegating a critical task to a fault-tolerant network are discussed.

* This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the first and third authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

1. Introduction

One of the most intriguing characteristics of biological neural networks is their extreme robustness with the ability to function even after severe damage. It has been shown that Artificial Neural Networks (ANNs) also exhibit some degree of "fault-tolerance," but in most cases the work did not explicitly focus on the fault-tolerance, which was demonstrated only as a side-effect [1, 20, 9, 13]. Fault-Tolerance is a qualitative, general term defined as the ability of a system to perform its function according to the specification in spite of the presence of faults in its subsystems. This definition is very unspecific and a system that is said to be fault-tolerant does not necessarily tolerate any number of faults of any kind in any of its subsystems. A specific way to quantify the fault-tolerance is to determine the *performance degradation* in the presence of certain faults in certain subsystems, given that some measure of performance exists.

Only relatively few studies in the literature are specifically concerned with the fault-tolerance of ANNs. Furthermore, the results are difficult to generalize because of the very different models and objectives. For example, Hinton and Shallice (1989) [10] "injected" faults into a backpropagation network trained to perform a particular linguistic task. They showed that the performance degradation of the network bears a qualitative resemblance to the degraded ability of neurological patients with a specific brain disorder. Petsche and Dickinson (1990) [17] used a special network architecture to investigate a self-repair mechanism that automatically activates spare nodes (neurons) if one of the nodes is "dead," i.e. permanently inactive ("stuck-at-0").

In this paper, we will investigate the time-continuous, recurrent ANN that was proposed by Hopfield in 1984 [11] to solve certain optimization problems. In the following, we will adopt the term "optimization networks" for these ANNs, which was coined by Tank and Hopfield [24]. Although optimization networks were initially applied to classical problems like the Traveling Salesman Problem, we are more interested in potential applications in real-time processing and control systems. For example, an optimization network implemented in analog hardware could perform a real-time scheduling or control task as a component of a hybrid system. If this is a critical task with high reliability requirements, then the allegedly "built-in" fault-tolerance of the neural network becomes a key factor. With such applications in mind, we will investigate the fault-tolerance of optimization networks and quantify the performance degradation in simulated "fault-injection" experiments. A broader goal is to gain insight into the principal character of the fault-tolerance of these networks and to explore the underlying cause.

The following two Sections of this paper contain a brief introduction to optimization networks and explain the principle of operation for two model problems, the Assignment Problem (AP) and the Traveling Salesman Problem (TSP). Section 4 introduces a performance measure that allows a meaningful assessment of how well the network actually solves

the AP and the TSP. Such a performance measure is a prerequisite for quantifying the performance degradation in the presence of simulated faults, which are "injected" into the network. Section 5 presents these results for the AP and TSP used again as model problems and discusses the cause and the effect of the observed fault-tolerance. Finally, Section 6 describes an application in which an optimization network is used for the real-time task allocation in a fault-tolerant, distributed processing system. The network is a critical component in this application and its fault-tolerance is an essential requirement for the operation of the system. The conclusion in Section 7 summarizes the main results and discusses the prospects of optimization networks for different application areas.

2. Optimization Networks

Figure 1 shows an optimization network in form of an electrical circuit model [12] with n interconnected amplifier units ("neurons") as the active circuit elements. The model allows resistive feedback from any output V_j to any input u_i with a resistor value R_{ij} or a conductance $T_{ij}=1/R_{ij}$, respectively. The current I_i can be used to provide an external input to the network. The nonlinear, sigmoidal *transfer function* that determines the relation between an input u_i and an output V_i is given by

$$V_i = \frac{1}{2}\left(1+\tanh\left(\frac{u_i-u_s}{u_0}\right)\right) = \frac{1}{1+\exp\left(-4\lambda\left(u_i-u_s\right)\right)} \quad ,$$
$$\lambda = \frac{1}{2u_0} = \frac{dV_i}{du_i}\Big|_{u_i=u_s} \quad . \tag{1}$$

The parameter λ denotes the slope of the transfer function at the inflection point $u_i=u_s$ and constitutes the maximum gain of the amplifier. The offset u_s is sometimes explicitly used as an additional parameter [4], but can be incorporated into the current I_i, which has also the effect of shifting the transfer function horizontally.

The feedback connections are described by positive and negative values for the *weight* T_{ij} of the connection between the output of unit j and the input of i. In an electronic circuit realization, $T_{ij}=1/R_{ij}$ can only be positive, and negative feedback requires the use of an additional output $-V_i$ for unit i ranging from 0 to −1. The intrinsic delay exhibited by any physical amplifier is modeled by an input resistance r_i and capacitance C_i. These are drawn as external components in Figure 1, so that the actual amplifier can be described as an ideal component with no delay.[1] A circuit analysis of the network in Figure 1 yields the "equations of motion"

[1] This is, however, an idealized model of a practical amplifier according to Smith and Portmann (1989) [21]. More realistic models might lead to instability of the system. (cp. also Marcus and Westervelt (1989) [15])

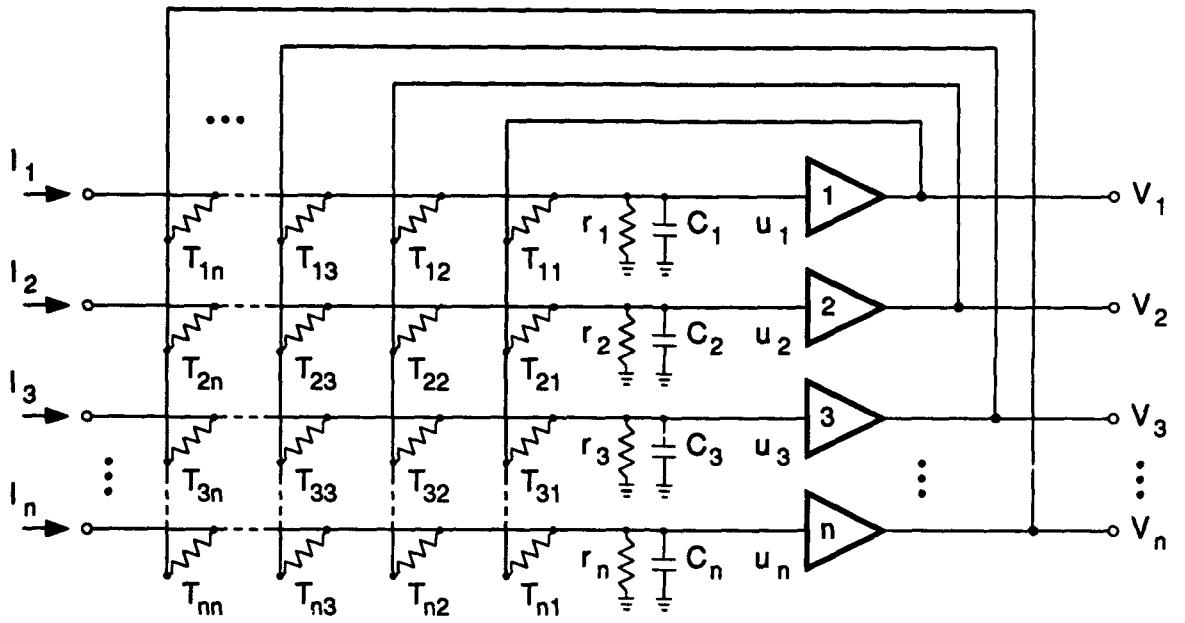

Figure 1. Circuit diagram of an optimization network according to Hopfield (1984) [11]. Note that negative feedback can be realized by connecting positive conductances T_{ij} to the negative output $-V_i$ of a unit (not shown in this figure).

$$C_i \frac{du_i}{dt} = -\frac{u_i}{R_i} + \sum_{j=1}^{n} T_{ij} V_j + I_i \tag{2}$$

that describe the time-evolution of the dynamical system. R_i represents the parallel combination of the input resistance r_i and all the weights $T_{ij}=1/R_{ij}$ connected to unit i according to

$$\frac{1}{R_i} = \frac{1}{r_i} + \sum_{j=1}^{n} T_{ij} \quad . \tag{3}$$

The product of R_i and C_i is often referred to as the time-constant τ_i of one particular unit i. An identical time-constant for each unit i would require $C_i=C$ and $R_i=R$ for all i. The latter condition might be difficult to achieve in practice if the parallel combination of the weights in (3) results in different values for each unit i. In this case, each individual value for r_i would have to be chosen in a way that compensates for these variations. It is also important to note that the time-constant τ_i describes the convergence of the input voltage u_i of unit i. Because of the potentially very high gain of the transfer function, the output V_i might saturate very quickly. Thus, even if the input u_i is still far from reaching its equilibrium point, the output V_i might already be saturated, and by observing only V_i it might *appear* as if the circuit had converged in merely a fraction of "its" time-constant τ_i.

Hopfield (1984) proved the stability of the nonlinear dynamical system (2) for symmetric connections ($T_{ij}=T_{ji}$). By introducing a Liapunov function [11], he showed that in the high-

gain limit ($\lambda \rightarrow \infty$) the stable states of the system correspond to the local minima of the quantity

$$E = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} T_{ij}V_iV_j - \sum_{i=1}^{n} V_iI_i \quad , \tag{4}$$

which Hopfield refers to as the "computational energy" of the system. This means that the dynamical system moves from an initial point in state space in a direction that decreases its energy (4) and comes to a stop at one of the many local minima of the energy function.

It has been shown [7] that the Liapunov function (4) for the system (2) is a special case of a more complex Liapunov function introduced by Cohen and Grossberg in 1983 [5], so that Equation (4) might not be considered as a new result in itself. Nevertheless, this does not diminish Hopfield and Tank's main contribution, which can be seen as their method of associating the equilibrium states of the network with the (local) solutions of an abstract optimization problem like the TSP. This method is briefly reviewed in the next section.

3. Solving Optimization Problems: Principle of Operation

The basic idea behind the operation of optimization networks can be stated as follows: If it is possible to associate the solutions of a particular optimization problem with the local *minima of the energy function* (4), *then the network* "solves" the problem automatically by converging from an initial state to a local minimum, which in turn corresponds to a (local) solution of the problem. This association requires a suitable problem representation, that is, an encoding of the problem by using the state variables V_i of the network. For example, the output V_i of a unit ranging from 0 to 1 can be used to represent a certain hypothesis that is true for V_i=1 and false for V_i=0. Different hypotheses can be encoded by different units and the hypotheses might have to satisfy certain constraints. If the final state of the network is supposed to represent a particular solution, it is usually required that the outputs V_i eventually converge to either 0 or 1 in order to obtain a decision. In this sense, the process of convergence with intermediate values $0<V_i<1$ could be interpreted as the simultaneous consideration of multiple, competing hypotheses by the network before it settles into a final state [23]. In the following, we will demonstrate the principle of operation for two model problems, the Assignment Problem (AP) and the Traveling Salesman Problem (TSP).

3.1 The Assignment Problem

The AP used for this example is a simple version, sometimes also called list matching problem, with the following specification. Given two lists of elements and a cost value for the pairing of any two elements from these lists, the problem is to find the particular one-to-one assignment or match between the elements of the two lists that results in an overall minimum cost. In order to distinguish clearly between the two lists, we use capital letters to

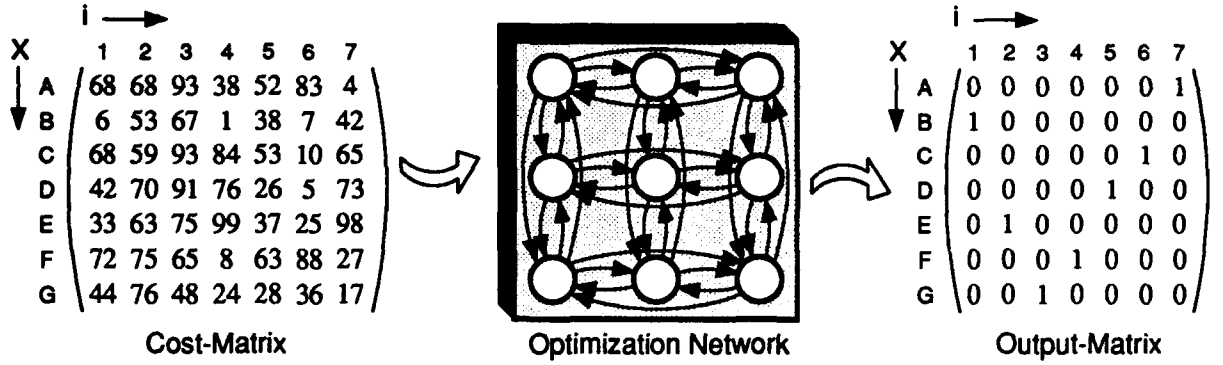

Figure 2. Exemplary cost-matrix for a 7×7 Assignment Problem and corresponding output matrix generated by the neural network. Here, the solution encoded by the output-matrix is optimal with an overall cost of c=165.

describe the elements of one list (i.e. X=A, B, C, etc.) and enumerate the elements of the other list (i.e. i=1, 2, 3, etc.). Additionally, we assume that the two lists contain the same number of elements n. A one-to-one assignment means that each element of X has to be assigned to exactly one element of i. The cost p_{Xi} for every possible assignment or pairing between X and i is given for each problem instance. This generic problem description has many practical applications, for example, the assignment of jobs i to processors X in a multiprocessor system by minimizing the cost of the communication overhead.

The AP as specified above can be represented by a two-dimensional, quadratic matrix of units, whose outputs are denoted by V_{Xi}. Thus, we can define V_{Xi} as a "decision"-variable, with $V_{Xi}=1$ meaning that the element X should be assigned to the element i, and $V_{Xi}=0$ meaning that the pairing between X and i should not be made. This way, a solution to the AP can be uniquely encoded by the two-dimensional matrix of the outputs V_{Xi} after all units converge to 0 or 1. Note that n^2 units are required to represent an AP with n elements per list. The constraints of the one-to-one assignment require that only one unit in each row and column converges to 1 and that all other units converge to 0. Thus, the outputs of the network after convergence should produce a *permutation matrix* with exactly one unit "on" in each row and column. Figure 2 illustrates this problem representation by showing the cost-matrix as the input for a particular problem instance and the output of the network after convergence. In this example, the output-matrix determines the assignment of elements A to 7, B to 1, C to 6, etc.

For the mapping onto the optimization network, the problem has to be expressed in the form of a quadratic function with minima representing the solutions. The "energy"-function

$$E_{AP} = \frac{A}{2}\sum_X\left(\sum_i V_{Xi} - 1\right)^2 + \frac{B}{2}\sum_i\left(\sum_X V_{Xi} - 1\right)^2 + \frac{C}{2}\sum_X\sum_i V_{Xi}(1 - V_{Xi}) + D\sum_X\sum_i p_{Xi}V_{Xi} \tag{5}$$

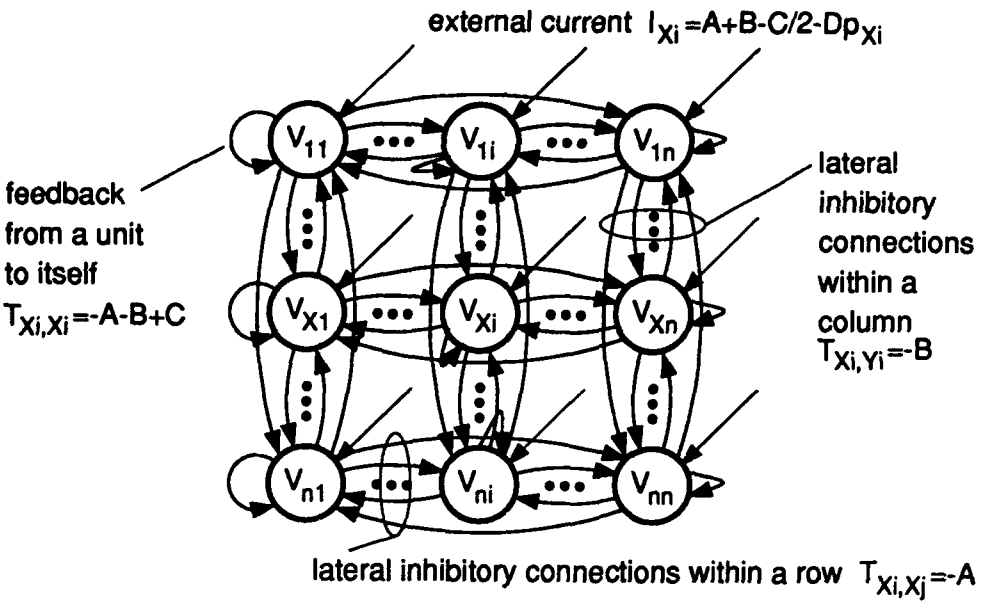

Figure 3. Schematic architecture of a two-dimensional neural network with the connectivity required to solve the Assignment Problem.

used by Brandt et al. (1988) [4] is such a function. The first two terms in (5) have minima if the sum over all outputs equals 1 for each row and each column, respectively. The third term has minima if all V_{Xi} are either 0 or 1 and, together with the first two terms, it enforces the constraints. The fourth term in (5) is simply the overall cost of a particular solution given the constraints are met. Furthermore, it is common to use constant factors A, B, C, and D as additional parameters in (5). These parameters have the effect of "weighting" the constraints and the cost-function and allow a fine-tuning of the performance as will be seen later.

The next step in mapping Equation (5) onto an optimization network is the derivation of the values for the connections and the external inputs. First, we have to extend the notation of the Liapunov function (4) to two dimensions:

$$E = -\frac{1}{2}\sum_X\sum_i\sum_Y\sum_j T_{Xi,Yj}V_{Xi}V_{Yj} - \sum_X\sum_i V_{Xi}I_{Xi} \quad . \tag{6}$$

By comparing Equations (6) and (5) it follows after some algebraic transformations that $E=E_{AP}$ if

$$\begin{aligned} T_{Xi,Yj} &= -A\delta_{XY} - B\delta_{ij} + C\delta_{XY}\delta_{ij} \\ I_{Xi} &= A + B - \frac{C}{2} - Dp_{Xi} \quad . \end{aligned} \tag{7}$$

Figure 3 sketches the resulting two-dimensional network architecture as a directed graph. With the specific values from Equation (7), the equations of motion for the AP become

$$C_{Xi}\frac{du_{Xi}}{dt} = -\frac{u_{Xi}}{R_{Xi}} - A\sum_{j} V_{Xj} - B\sum_{Y} V_{Yi} + CV_{Xi} + A + B - \frac{C}{2} - Dp_{Xi} \quad . \tag{8}$$

3.2 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) was the first example chosen by Hopfield and Tank (1985) [12] to demonstrate how a neural network could be used to solve optimization problems. The task of the traveling salesman is to visit n cities, once each, in a closed tour in such a way that the overall length of the tour is minimal. The TSP is a classical, NP-complete optimization problem [6], for which no algorithm exists that could find a (global) solution in polynomial time. Hopfield and Tank's TSP example achieved such prominence because it was one of the first examples of a neural network "solving" a problem that is intractable for conventional computers. However, as we will discuss later, the TSP was meant and should be regarded as an *example* only, and does not suggest that a general method has been discovered that solves NP-complete optimization problems.

The problem representation for the TSP is similar to the AP and requires a two-dimensional network with outputs V_{Xi}. The difference is that the first index (X) now denotes a city, and the second index (i) describes the order in which a city is visited along the tour. The representation of a problem with n cities requires a quadratic matrix of n^2 units whose outputs V_{Xi} should converge to binary values. We define V_{Xi}=1 as the decision that city X should be on the *i*th position of the tour. Conversely, V_{Xi}=0 determines that city X should not be on the *i*th position. The requirement of the TSP that each city has to be visited exactly once can be rephrased such that each city can be in only one position of the tour and each position can be occupied by only one city. Thus, the constraints are met if the outputs of the network converge to a permutation matrix with only one "1" in each row and column. This means that the mathematical expression of the constraints in form of a quadratic function is identical to the one derived for the Assignment Problem. However, the problem representation has a 2n-fold degeneracy because there exist n matrices for each of the two directions of traversal that encode the same tour.

Except for a different cost-function, the energy-function for the TSP is identical to the AP and can be written as [4]

$$E_{TSP1} = \frac{A}{2}\sum_{X}\left(\sum_{i} V_{Xi} - 1\right)^2 + \frac{B}{2}\sum_{i}\left(\sum_{X} V_{Xi} - 1\right)^2 + \frac{C}{2}\sum_{X}\sum_{i} V_{Xi}(1 - V_{Xi})$$

$$+\frac{D}{2}\sum_X\sum_Y\sum_i d_{XY}V_{Xi}\left(V_{Y,i+1}+V_{Y,i-1}\right) \quad . \tag{9}$$

The fourth term in (9) represents the cost function, which is simply the length of the overall tour scaled by the parameter D. The mapping of (9) onto the Liapunov function of the network results in the following network parameters

$$\begin{aligned} T_{Xi,Yj} &= -A\delta_{XY} - B\delta_{ij} + C\delta_{XY}\,\delta_{ij} - Dd_{XY}\left(\delta_{j,i+1}+\delta_{j,i-1}\right) \\ I_{Xi} &= A + B - \frac{C}{2} \quad . \end{aligned} \tag{10}$$

The principal difference between the TSP connectivity in (10) and the AP connectivity in (7) is that the TSP cost-function is encoded by the connections and not by the external current. The architecture of the TSP network is identical to the AP network as illustrated in Figure 3, except that the TSP network has a constant I_{Xi} and the additional connections $T_{Xi,Yj}=-Dd_{XY}(\delta_{j,i+1}+\delta_{j,i-1})$.

The equations of motion that describe the dynamics of the TSP network are

$$\begin{aligned} C_{Xi}\frac{du_{Xi}}{dt} = &-\frac{u_{Xi}}{R_{Xi}} - A\sum_j V_{Xj} - B\sum_Y V_{Yi} + CV_{Xi} \\ &- D\sum_Y d_{XY}\left(V_{Y,i+1}+V_{Y,i-1}\right) + A + B - \frac{C}{2} \quad . \end{aligned} \tag{11}$$

Originally, Hopfield and Tank proposed a different energy-function for the TSP, which uses an alternative formulation to enforce the constraints. Their original TSP energy-function [12] is

$$\begin{aligned} E_{\mathrm{TSP2}} = &\frac{A}{2}\sum_X\sum_i\sum_{j\neq i} V_{Xi}V_{Xj} + \frac{B}{2}\sum_i\sum_X\sum_{Y\neq X} V_{Xi}V_{Yi} + \frac{C}{2}\left(\sum_X\sum_i V_{Xi} - n\right)^2 \\ &+\frac{D}{2}\sum_X\sum_Y\sum_i d_{XY}V_{Xi}\left(V_{Y,i+1}+V_{Y,i-1}\right) \quad . \end{aligned} \tag{12}$$

The mapping of (12) onto the Liapunov function (6) results in the values

$$\begin{aligned} T_{Xi,Yj} &= -A\delta_{XY} - B\delta_{ij} + (A+B)\,\delta_{XY}\,\delta_{ij} - C - Dd_{XY}\left(\delta_{j,i+1}+\delta_{j,i-1}\right) \\ I_{Xi} &= nC \end{aligned} \tag{13}$$

and in the corresponding equations of motion

$$\begin{aligned} C_{Xi}\frac{du_{Xi}}{dt} = &-\frac{u_{Xi}}{R_{Xi}} - A\sum_{j\neq i} V_{Xj} - B\sum_{Y\neq X} V_{Yi} - C\sum_Y\sum_j V_{Yj} \\ &- D\sum_Y d_{XY}\left(V_{Y,i+1}+V_{Y,i-1}\right) + Cn \quad . \end{aligned} \tag{14}$$

The main difference between Hopfield and Tank's original formulation (12–14) and the modification (9–11) is the "global inhibition" term –C in Hopfield and Tank's equation (13) as well as an external current term that depends on the problem size n. Although both approaches seem to be equivalent in the sense that both enforce the convergence to a permutation matrix while using an identical cost-function, their performance turns out to be considerably different. In trying to recreate Hopfield and Tank's original results, many people have reported poor results, that is, either the network failed completely to converge to a valid tour (permutation matrix) or the solution was clearly far from the global optimum [26, 25, 8]. These problems do not occur when the alternative formulation of the energy function (9) is used [4]. However, the performance still depends strongly on the parameter values, on the initial values, and on the cost-function of the underlying city-distribution.

4. Performance Assessment

The performance assessment would not be an issue if the network simply found the global solution all the time. In fact, this would imply a solution to the NP-completeness problem. However, the network usually converges to local minima and produces good but "suboptimal" solutions. Then the question becomes "how good is good?" and the need for a performance measure arises. One obvious measure of performance is, of course, the resulting cost-value after convergence, given that the network converged to a valid solution. For the TSP, this is simply the distance of the tour, and the smaller the distance the better the network performs. Unfortunately, the performance of a given network varies considerably for different problem instances (data sets), for different problem sizes, for different network parameters, and in the case of the TSP, also for different initializations of the network. This variation impedes a meaningful, general performance assessment if only one or two example problems are considered, because it is always possible to "fine-tune" the network parameters for a particular problem instance.

Therefore, it is necessary to generate a representative number of examples that allows a statistically meaningful statement to be made about the *average* performance. Furthermore, some reference frame is needed for the comparison of the network results, because just the average over the cost-values is generally not sufficient. The simplest reference for a comparison is the average cost-value of a "random guess," that is, the average or expected value of the distribution of all possible answers for a particular problem instance. A performance assessment based on the estimated distribution has led to statements in the literature that, for example, a solution is approximately among the 10^8 best out of 4.4×10^{30} possible solutions (Hopfield and Tank, 1985 [12]), or that 92% of the solutions are among the best 0.01% of all solutions (Tagliarini and Page, 1987 [23]). While this gives some impression of the performance, it can hardly be considered a practical measurement.

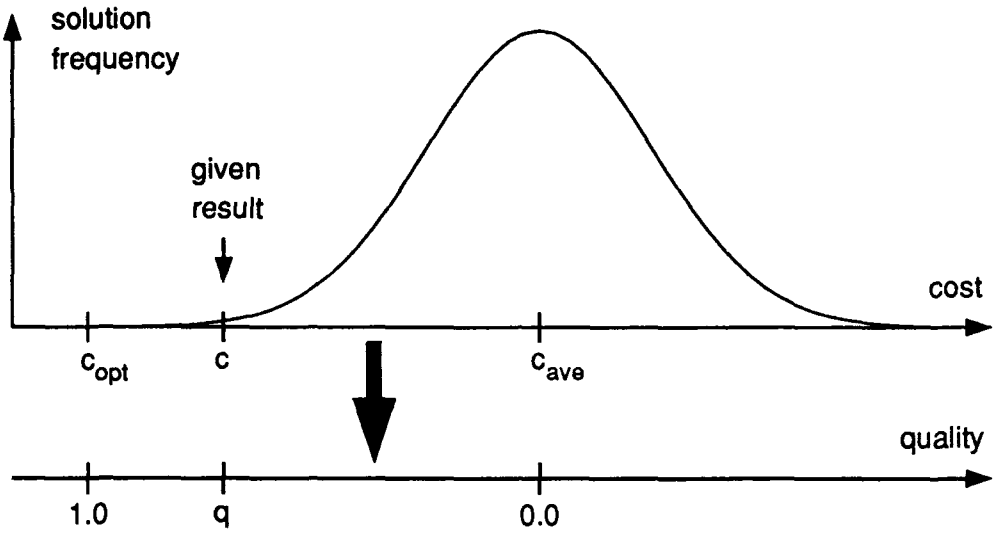

Figure 4. Definition of a solution quality q by mapping the absolute values c, c_{opt}, and c_{ave} onto a normalized scale.

What is needed is a performance measure that can give an answer to the following questions:

- What is the effect of a parameter variation or a modification of the energy function on the performance?
- How good is the solution with respect to the global optimum or the best known answer?
- How does the performance change with problem size?
- With respect to fault-tolerance, how does the performance degrade under the presence of (simulated) faults?
- What is the performance difference of two networks solving two different problems, that is, are there problems that are "easier" for the network to solve?

Our approach to the performance assessment is based on the fact that the distribution of all possible answers for every instance of an optimization problem can be characterized by two values, the global optimum (minimum cost) c_{opt} and the average cost value c_{ave}. With c denoting the cost value of a given result derived by the network, the relation between c, c_{opt}, and c_{ave} can be used as a performance measure. By mapping those absolute values onto a normalized scale as illustrated in Figure 4, we define the *solution quality q* as

$$q = \frac{c_{ave} - c}{c_{ave} - c_{opt}} \quad . \tag{15}$$

Thus, the solution quality has a value q=1 if $c=c_{opt}$ and q=0 if $c=c_{ave}$, with $0<q<1$ for $c_{ave}>c>c_{opt}$.

Traveling Salesman : Different Approaches	Problem Size n (Number of Cities)				
	10	20	30	50	100
1.) Original Method of Hopfield and Tank	0.905 (0.15)	0.903 (0.11)	0.851 (0.02)	- (0.00)	-/-
2.) Modified Method of Brandt et al. (1988)	0.829 (1.00)	0.816 (1.00)	0.830 (1.00)	0.852 (1.00)	0.902 (1.00)
3.) Brandt et al. (1988), different parameters	0.936 (0.98)	0.926 (0.97)	0.923 (0.84)	0.913 (0.58)	0.927 (0.18)

Table 1. TSP solution quality q and proportion of valid solutions (in parentheses) for different problem sizes and solution methods.

Obviously, the calculation of q requires the knowledge of the two reference values c_{opt} and c_{ave} for each problem instance (e.g. for each city distribution of the TSP). Obtaining values for c_{ave} is usually no problem since it only requires a sufficient number of random trials. In case of the TSP, for example, a random but valid tour is generated repeatedly and the resulting tour lengths are averaged to obtain c_{ave}. The fact that we have to know the global optimum c_{opt} appears to be a paradox at first glance and one might ask why we would use an ANN to solve a problem for which the best possible solution is already known. The answer is, of course, that we want to *test* the network by using well-known *model problems* and for such a test it is reasonable to compare the results of a new method (i.e. ANNs) to the results of the best existing method. In fact, in almost all cases, where ANNs have been applied to optimization problems, there are conventional algorithms readily available to provide values for c_{opt}. For NP-complete optimization problems like the TSP, for which the global optimum is generally unknown, the best available heuristic method like the Lin-Kernigham algorithm [14] can be used as a reference. If c_{opt} is not the global optimum and should it happen that the network generates a better answer, then the event $c<c_{opt}$ is reflected by a solution quality $q>1$. Conversely, the value for q becomes negative if the solution of the network is worse than the random average ($c>c_{ave}$). Thus, the normalized solution quality is independent of a particular problem instance and of the problem size.

In the following, we will demonstrate the use of the defined solution quality to assess and compare the performance for the two model problems, the TSP and the AP. In order to get statistically relevant results for the TSP, we generated a *test-set* containing 10 different city distributions for each problem size (n=10, 20, and 30) and 5 different distributions for n=50 and 100. Each city distribution was generated by placing the cities randomly on a unit square according to a uniform probability distribution. The values for c_{ave} were obtained by averaging over 10^5 random trials for each city distribution. The Lin-Kernigham algorithm [14] was used to generate 5 answers for each city-distribution and the best result

was chosen as c_{opt}. Since the network performance varies considerably for different random initializations, 10 different initializations were used for each city distribution of size 10 to 50, and 5 initializations for n=100. Thus, a single sweep through the test-set requires 375 simulation runs and the value for q was calculated after each run. The average values for q are shown in Table 1 for different approaches and problem sizes.

There is also the possibility that the network will not converge at all to a valid solution because it has gotten stuck in a local minimum ("spurious attractor") that does not correspond to a permutation matrix. Since this event is not reflected by the solution quality, we also show in Table 1 the proportion of runs with valid solutions. The average value for q includes only runs that produced valid solutions. In an attempt to recreate Hopfield and Tank's original results, we performed a run of the test set using their original equations (12–14) with the parameters A=B=500, C=200, D=500, λ=25, and u_s=0 as described in [12]. Furthermore, Hopfield and Tank used an additional constant term for the external current according to I_{Xi}=C(n+5)=200n+1000, which effectively shifts the transfer function. They also used the initialization $V_{Xi}(t=0)=1/n+\delta$ where δ is a small random number [12].

The equations of motion (14) were solved by Euler's method with time-steps Δt between 10^{-5} and 10^{-6}. A larger Δt can cause numerical errors and results that do not reflect the actual behavior of the system. The first row in Table 1 shows the results of our simulation that confirm the reported difficulties [26, 8, 4] in using Hopfield and Tank's original equations. Even for n=10 cities only 15% of the runs converged to a valid solution and since none of the 50–city cases produced a valid answer we did not even attempt to solve a 100–city problem.

Although we experimented extensively with parameter variations, we did not find a set of parameters that improves the performance significantly. However, it is possible to "fine-tune" the parameters for *one particular* city-distribution to obtain quite impressive results. Unfortunately, the same parameters usually produce invalid or poor results for other city-distributions. This characteristic has led to some confusion in the literature with performance claims based on specific examples that were difficult to reproduce and did not hold in general [26]. This also demonstrates the importance of an *average* performance assessment over many examples. Since Hopfield and Tank's original equations (12–14) are not the only way to express the problem, we tried different modifications [19, 18] and obtained the best results with the approach published by Brand et al. (1988) [4] that is described in Section 3.3. By using Brandt's energy equation (9) and his original parameters A=B=2, C=4, D=1, λ=2.5, and u_s=0.5, we obtained the results shown in the second row of Table 1. An additional difference of Brandt's approach is an initialization in the center of the hypercube with $V_{Xi}(t=0)=0.5+\delta$ and a random variable δ uniformly distributed in the range $-10^{-6}\leq\delta\leq10^{-6}$. Because of the lower gain and smaller values of the parameters, we could use the value Δt=0.1 to solve the equations of motion (11).

As shown in the second row of Table 1, this modified energy function produced consistently valid tours across the full range of problem sizes. However, the average solution

quality was lower than the valid cases of Hopfield and Tank's results. We tried different parameters for Brandt's energy equations to improve the quality. The results for A=B=5, C=2, are D=3 are listed in the third row of Table 1. The parameters for the transfer function and the initialization are the same as in the previous case, except that we used a $\Delta t=5\times10^{-3}$. We can see that the average quality has indeed been improved, but at the price of occasional invalid answers whose frequency increases with the problem size. There is a fundamental tradeoff between obtaining consistently valid (but sometimes poor) answers for a large number of different problem instances and very good answers for a small number of instances. One obvious and extreme case of this tradeoff is setting D=0, which cancels the cost-function and reduces the problem to pure constraint satisfaction. Then we would always expect valid answers, but with an average quality of q=0. The underlying problem with the TSP is the quadratic cost-function that maps the problem-specific distance values multiplied by the parameter D onto the connections, where they are added to the values that enforce the constraints as in (10) or (13). Qualitatively speaking, large distance values in an extreme problem case or a large factor D might "override" the connectivity values that enforce the constraints and thus interfere with the convergence to a valid solution.

This problem does not occur with the Assignment Problem because the energy function for the AP (5) maps the problem-dependent cost values to the external current (7) and not to the connection values. This is actually the only difference between the AP- and the TSP-network, as far as the architecture is concerned, and makes a performance comparison between the problems especially interesting. As before, we generated a test-set of 10 problem instances for each size of 10, 20, 30, 50, and 100 elements. The cost values were randomly generated with a uniform distribution between 0 and 1. The AP as defined here is not an NP-complete problem and there exist relatively simple and fast algorithms that find the global solution. We used such a textbook algorithm [22] to obtain values for c_{opt} and generated the average values c_{ave} from 10^5 random solutions for each problem instance. The first row of Table 2 shows the simulation results for the parameters originally used by Brandt et al. [4] with the additional values λ=2.5, u_s=0.5, Δt=0.05, and the initialization $u_{Xi}(t=0)$=0. The other two rows show the effect of parameter modifications and here the values λ=25, u_s=0, $\Delta t=5\times10^{-5}$ were used with the same initialization. In contrast to the TSP, no random bias in the initial values is required for the AP; in fact, the network converges to the same solution despite some small random noise. This simplifies the performance assessment considerably, because we now need only one simulation run for each problem instance.

A comparison between Table 1 and 2 reveals a striking difference between the TSP- and the AP-results. For the AP, none of the runs failed to converge to a valid solution and, moreover, the solution quality is excellent. For the parameter sets 2.) and 3.) in Table 2, the network actually found the global optimum in most cases or generated an answer extremely close to it. We can conclude that the "non-interference" of the cost-values with the connection-values that enforce the constraints is the cause for the enormous performance

Assignment Problem: Different Parameters	Problem Size n (Number of Elements)				
	10	20	30	50	100
1.) A=B=2, C=2, D=1	0.988 (1.0)	0.960 (1.0)	0.975 (1.0)	0.978 (1.0)	0.987 (1.0)
2.) A=B=200, C=20, D=50	1.0 (1.0)	0.999 (1.0)	0.999 (1.0)	0.998 (1.0)	0.998 (1.0)
3.) A=B=200, C=3, D=50	1.0 (1.0)	0.999 (1.0)	1.0 (1.0)	1.0 (1.0)	0.999 (1.0)

Table 2. AP solution quality q and proportion of valid solutions (in parentheses) for different problem sizes and parameters.

difference. Thus, the distinction between a quadratic and a linear cost function becomes an important classification which helps to identify problems that are more suitable to an ANN-implementation. The demonstrated ability to compare the results of two different optimization problems proves the versatility of the solution quality as a performance index and justifies the additional effort needed to obtain values for c_{opt} and c_{ave}.

There is another aspect to the comparison between optimization networks and conventional algorithms, which is the time it takes to solve a problem of a particular size. For example, it takes more than one day of CPU time on a VAX 780 to *simulate* the neural network solving a single 100-city problem. This is actually not surprising because the simulation involves the numerical solution of 10^4 ODEs for several thousand iterations. However, the Lin-Kernigham algorithm provides a (usually much better) answer in about 3 minutes. Furthermore, 100 cities are not even considered an "interesting" problem size for the TSP. Although an analog hardware implementation of the neural network might solve the same problem in milliseconds, the need for a VLSI chip with 10^4 Operational Amplifiers to solve a 100–city TSP is truly questionable. Thus, we do not think that large-scale, classical or NP-complete optimization problems are suitable applications for optimization networks other than as examples or model problems. However, there are certain small-scale, special purpose, real-time control problems that could benefit from the key characteristics of an ANN hardware implementation: speed, low weight and power consumption, and "built-in" fault-tolerance. Therefore, our actual objective is not to compete with conventional methods in solving classical optimization problems, but to investigate the fault-tolerance of the network for special purpose applications. The above performance assessment is a prerequisite for this investigation.

5. Fault-Tolerance

It is possible to distinguish between two different characteristics, which we might call

static fault-tolerance and *dynamic fault-tolerance*. A system with static fault-tolerance does not react in any special way to compensate for the effect of internal failures, whereas a dynamically fault-tolerant system reorganizes its resources to counteract the fault-effects actively. An example for the latter case is adaptation or retraining after internal faults [1, 20] or the self-repair mechanism proposed by Petsche and Dickinson (1990) [17]. Generally, it is more difficult to achieve the same degree of robustness with static fault-tolerance because no repair or reconfiguration is possible. Since optimization networks are "hard-wired" and do not adapt or learn, they can exhibit only static fault-tolerance. Thus, we will "inject" simulated faults into the network and observe the performance degradation by using the defined solution quality for the TSP and the AP. A study that is related to our approach was performed by Belfore and Johnson (1989) [3] who also investigated the effect of faults in an optimization network that solves the TSP. However, they used only a single 6–city distribution with single node faults in their simulations, which is insufficient to draw any statistically meaningful conclusions as we will show below.

According to Figure 1, there are only two different components in a hardware implementation of an optimization network, the "neuron" or active element in the form of an operational amplifier and passive interconnections in the form of resistors. In the following, we will first consider two types of faults of the active elements that correspond to the highest failure rate. These are commonly called "stuck-at-1" or "stuck-at-0" faults and occur if the output of a unit (amplifier) is permanently pulled to the highest potential or to the lowest (ground) potential, respectively. The fault-locations are randomly selected with one important exception: *we do not allow two stuck-at-1 faults to occur within the same row or column*. The reason is that such an event would automatically preclude a valid solution, since the permutation matrix allows only one "1" in each row and column. In simulating multiple faults, we study a succession of either stuck-at-1 or stuck-at-0 faults, but not a mixture of both types. We use the same locations for stuck-at-1 and stuck-at-0 faults, in order to compare the effect of a different fault type. Otherwise it would not be possible to tell whether different results are caused by the different locations or by the different fault types. This means that the above exception is also valid for stuck-at-0 faults although two or more stuck-at-0 faults in the same row or column do not necessarily interfere with a valid solution.

Before we present the results of our large-scale simulations, we want to illustrate the impact of stuck-at-1 faults for two examples. Figure 5 demonstrates the effect of 4 stuck-at-1 faults simultaneously present in a network solving a 10-city TSP. The network parameters are those that produced the results in the second row of Table 1 and are listed in the previous Section. For comparison, Figure 5a shows a good but suboptimal solution of length 3.08 for a fault-free network. The locations of the 4 injected faults are visible after the initialization in Figure 5b. In Figure 5c it can be seen that the network still converges to a solution; however, the resulting tour of length 3.77 is clearly worse than in the fault-free case. In order to understand these results, it is necessary to recall the "meaning" of a fault in this

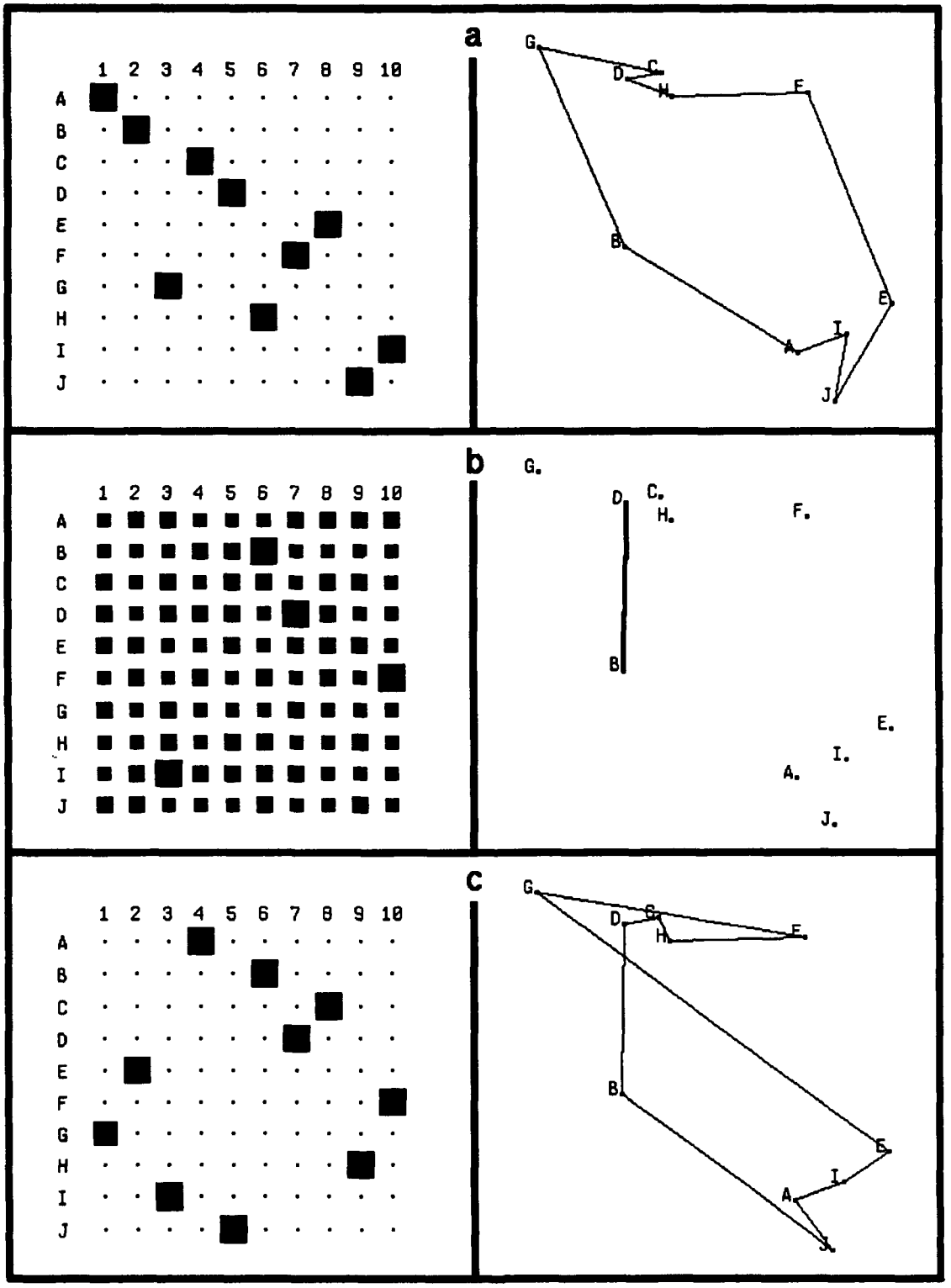

Figure 5. Solution of a 10–city problem by a network without any faults (a), new initialization of the network now with 4 stuck-at-1 faults (b), and solution under the presence of faults (c). Note that the two faults in adjacent columns predetermine a link between the cities B and D.

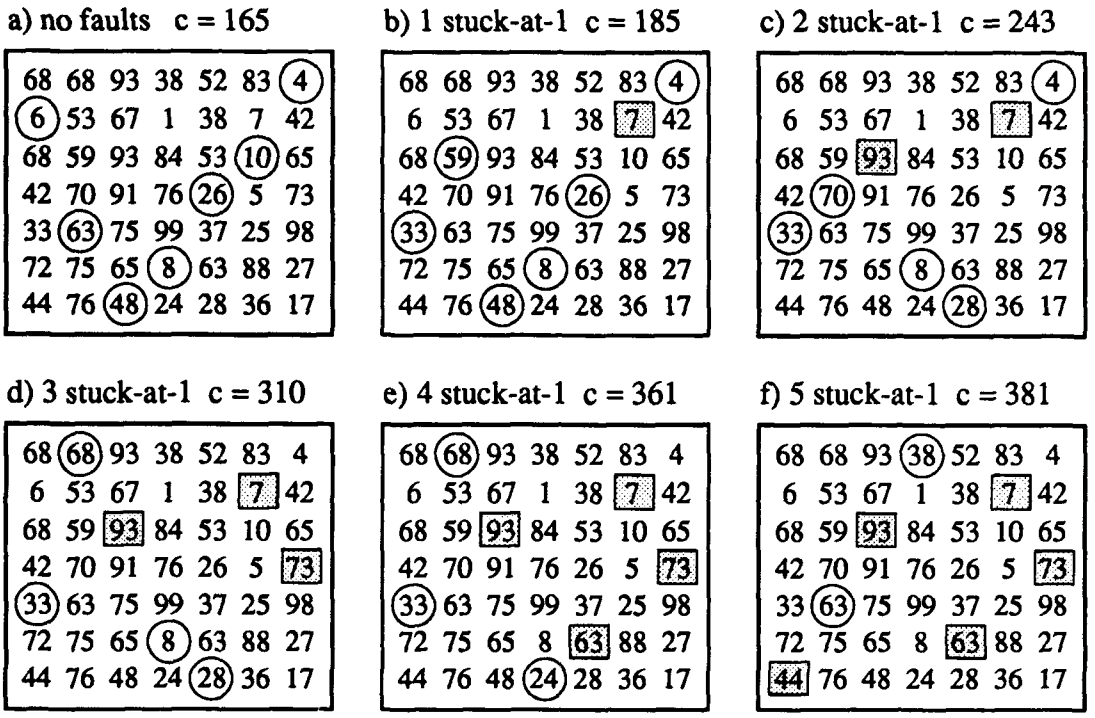

Figure 6. Effect of up to 5 multiple stuck-at-1 faults on a network solving an Assignment Problem of size n=7. Shown is the cost-matrix with the circled elements indicating the network solutions (neurons that converged to "1") and the shaded squares indicating the fault locations. Note that b)-f) are still optimal solutions under the additional constraints imposed by the faults.

context. Since we interpret the neuron output as a decision about the position of a city on a tour, a stuck-at-1 fault represents such a decision and thereby predetermines a part of the overall tour. Because of the degeneracy of the TSP problem representation, a single stuck-at-1 fault does not constrain the network at all since the absolute position of a city does not matter. The effect of two simultaneous faults is immediately obvious if the two faults occur in adjacent columns. As shown in Figure 5b, such an event predetermines a link between two cities because the cities are in successive positions on the tour. Figure 5c shows how this imposed "link" affects the overall tour.

Surprisingly, this predetermination of parts of a tour by the injected faults does not necessarily lead to a performance degradation. Since the network usually finds a suboptimal solution in the fault-free case, it is conceivable that a "lucky" combination of fault-locations leads to a tour that is actually better than one without any faults. While these events are rare, we could observe occasional improvements under the presence of multiple faults. Stuck-at-0 faults play a less prominent role because they only *preclude* a city from being in a certain position instead of predetermining it. Thus, the network has even more ways to "work around" those faults and we would expect a minimal impact even for multiple stuck-at-0 faults.

Figure 6 shows the effect of injected stuck-at-1 faults on a network solving the Assignment Problem. The parameters used for this example are those listed in the second row of Table 2. The solution shown in Figure 6a represents the global optimum. Thus, if the best answer is derived under fault-free conditions, any fault can only decrease the performance. Because the AP problem representation does not have the degeneracy like the TSP, even a single stuck-at-1 fault precludes a convergence to the global solution. Figure 6b–f illustrates how the multiple fault-locations marked by the shaded squares become part of the solutions and how the network converges to accommodate these constraints.

We analyzed the network solutions in Figure 6b-f by using our conventional algorithm and by taking the faults into account as additional constraints to the problem. Interestingly, the network arrived at the same results, which means that it still found the new "global" optimum under these fault-conditions. Thus, we could define a *conditional performance measure* by viewing the faults as constraints to the problem and assessing the network performance accordingly. Although we can see the obviously unavoidable performance degradation in absolute terms, the conditional performance of the AP network is still optimal. As with the TSP, stuck-at-0 faults preclude a particular solution and have no effect at all on the AP unless the fault location coincides with an active unit that is part of the solution. In this case, we have observed the same phenomenon that the network treats the fault as an additional constraint and converges to the "best possible" solution.

Although the above examples provide some (qualitative) insight into the fault-tolerance characteristics, it is still necessary to substantiate this impression by large-scale simulations in order to obtain more rigorous results. We used the test-set of problem instances as defined in the previous section and the same parameters that correspond to the results in the second row of Tables 1 and 2. Only these parameter values were used for the TSP because we regard the consistent convergence to a valid solution in the fault-free case as a prerequisite for any fault-injection experiments. Figure 7 shows the results for different problem sizes. The results confirm our conjecture that stuck-at-0 faults have no effect for the AP and practically no effect for the TSP. In case of the TSP, the injected faults "override" the random initialization and the network converges without or independent of any initial bias to the same solution. Stuck-at-1 faults result in an almost linear performance degradation for the AP, while the redundancy of the TSP problem representation is reflected in a relatively slower performance decrease as the number of faults increases. When the number of stuck-at-1 faults approaches the number of cities or elements, the performance for both the TSP and the AP approaches zero as in Figure 7a, which corresponds to the random average. This is because the randomly selected fault locations eventually predetermine a random tour. Most importantly, none of our simulations failed to converge to a valid tour because of one or more injected faults.

In another experiment, we studied the effect of connection faults on the performance of an optimization network. Although the failure rate of a simple resistive connection is orders

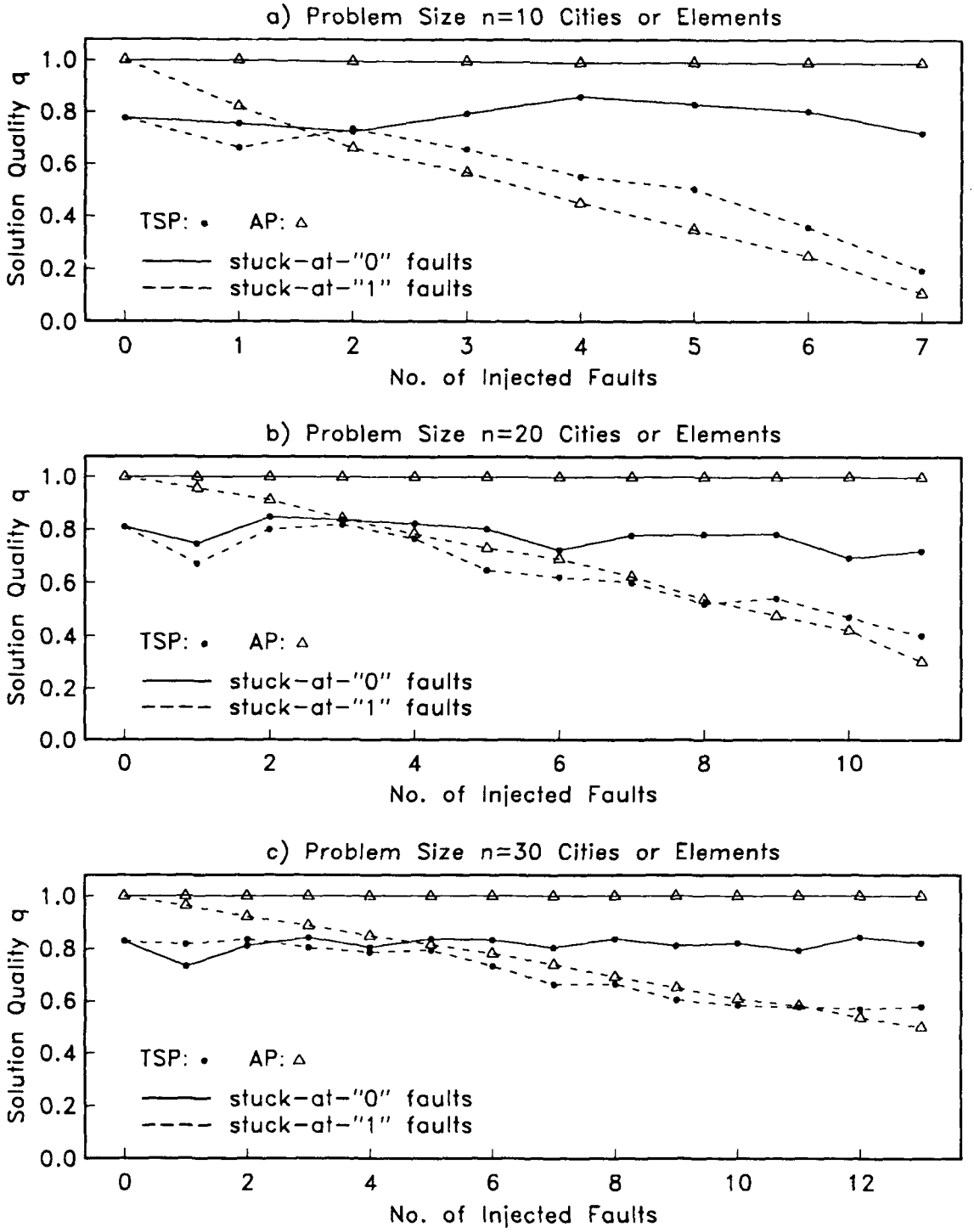

Figure 7. Performance degradation of an ANN solving the Traveling Salesman Problem (TSP) and the Assignment Problem (AP) after injections of stuck-faults for different problem sizes. The values are averages over 10 different problem instances for each size with additionally 10 different random initializations each for the TSP.

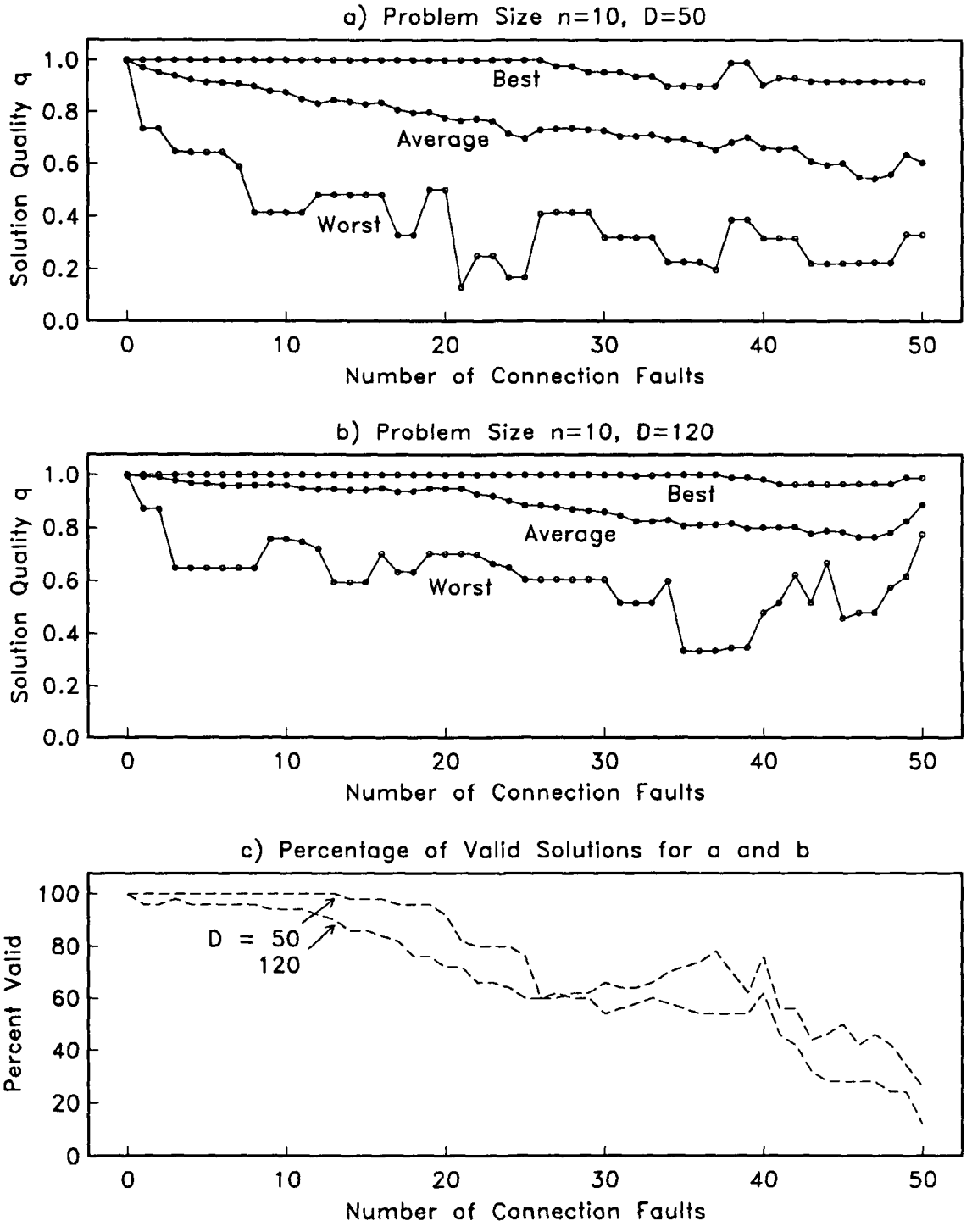

Figure 8. Performance degradation of an ANN solving the Assignment Problem (AP) after multiple connection failures (open connections). The values in a) and b) are the best, worst, and average performance of 50 different problem instances and the values in c) indicate how many out of the 50 runs for each fault-scenario converged to a valid solution.

of magnitude less than that of an operational amplifier, the large number of connections (e.g. $2n^3$–$2n^2$ connections for an n-element AP compared to n^2 neurons) increases the overall probability of such a fault. The failure of a connection with the resistance R leads either to a short circuit (R=0) or to an open connection ($R = \infty$). Because the failure rate of a connection short circuit is far less than the rate of an open connection, we simulated only the latter fault-type. In order to limit the number of required simulations we only used a network solving the AP for this experiment, because this network exhibited the best performance and greatest fault-tolerance in our previous studies.

Figure 8 shows the resulting performance degradation of an ANN solving a 10–element AP for up to 50 simultaneous open connections. The parameters for the AP-network are the same as in the previous fault-injection runs. The locations for the connection faults were randomly selected. For each fault-scenario we ran 50 different problem instances and Figure 8a-b shows the average as well as the worst and the best performance for the two different values of the parameter D=50 and D=120. The parameter D is a factor multiplied by the cost values according to Equation (7) and a large value for D enforces solutions with better quality. This is reflected by Figure 8b which shows a better average quality as well as a lower variation in the quality of the best and the worst solution compared to Figure 8a. This high variation in Figure 8a is again a reminder how much the results depend on the chosen problem instance and that the study of a single instance as in [3] can lead to grave misinterpretations.

Although the performance results suggest that a higher value for D would be desirable, there is a tradeoff shown in Figure 8c. Surprisingly, while none of the "stuck-at" faults led to an invalid solution, we do observe invalid solutions for some problem instances after a certain number of open connections. Figure 8c shows the percentage of valid solutions and it can be seen that a lower value for D tolerates more faults before the first case of an invalid solution occurs. We have already seen this tradeoff between consistently valid and high quality solutions in the fault-free cases of Section 4 and it is very interesting to observe that the same effect plays an important role with respect to the fault-tolerance. Because an invalid solution is the worst case and equivalent to a total system failure, a small value for D is obviously preferable, especially since it does not affect the fault-free performance at least for the cases shown in Figure 8a and b. However, for a value D>120 we could also observe some invalid results in the fault-free case. This shows that the "quality-validity tradeoff" is a general phenomenon and that connection faults only increase the likelihood of invalid solutions.

In summary, we have demonstrated that optimization networks exhibit a surprising degree of fault-tolerance, which is achieved without the explicit use of redundant components. Because the fault-tolerance characteristics are inseparable from the functional characteristics, we can say that the fault-tolerance of the ANN is "built-in" or *inherent*. However, when we make a statement about the fault-tolerance, we implicitly assume a failure condition or *fail-*

ure criterion of the system, which is the threshold below which it can no longer perform its function according to the specification. For example, consider the AP-network that always generates the global optimum under fault-free conditions. If we specify this as the only acceptable performance level, than any stuck-at-1 fault that causes the network to generate a good but suboptimal answer is not acceptable and, with respect to this fault-type, the network is not fault-tolerant at all. On the other hand, if we specify a solution quality of 0.8 as the acceptable performance threshold, then an AP-network of size n=30 can tolerate (on the average) 5 stuck-at-1 faults and an even larger number of stuck-at-0 or connection faults. Thus, the degree of fault-tolerance depends on our definition of acceptable performance.

The main reason that optimization networks are interesting from a fault-tolerance perspective is that they exhibit a *graceful performance degradation* and that they do not have a critical component. Most conventional systems are either fully operational or break down completely if a single fault occurs in a critical component or subsystem. Furthermore, most neural networks have critical components and are therefore not truly fault-tolerant. Consider, for example, a feedforward (backpropagation) network that is trained as an autopilot to control the altitude of an aircraft and has a single neuron in its output layer whose analog value represents the control variable. While this network might tolerate multiple connection faults or faults of its hidden units, a single stuck-at-0 or stuck-at-1 fault at the output neuron would lead to a total system failure. Because of the critical component, such a network is, at least in the strong sense of the definition, not fault-tolerant at all.

The above discussion suggests an application domain for optimization networks, where it is not necessarily important to generate the best possible solution to an optimization problem, but where a "reasonably" good answer has to be obtained fast and reliably. In the next section we present an example of such an application with the network performing a critical real-time task as a component of a fault-tolerant multiprocessor system.

6. Application of an ANN for the Task Allocation in a Distributed Processing System

In the following we will investigate the application of an optimization network in the context of a distributed processing system that operates under hard real-time constraints and has to meet very high reliability requirements. An example of such a system is the Software-Implemented Fault-Tolerance (SIFT) computer used by NASA as an experimental vehicle for fault-tolerant systems research [16]. The SIFT architecture can accommodate up to eight processors in a fully distributed configuration with a point-to-point communication link between every pair of processors. It can be used, for example, to execute real-time flight control tasks as part of an aircraft control system. Because the system operates in a distributed fashion, each processor executes a certain number of tasks according to a predetermined task-to-processor allocation table. The architecture achieves an extreme fault-

tolerance by its capability to detect and to isolate possible hardware faults. The isolation of a defective processor requires a reconfiguration of the system and a reallocation of all tasks among the remaining processors. Thus, it is not the initial task allocation, but the reallocation of tasks after a processor failure, that is time-critical and has to be performed by a highly reliable mechanism. The use of look-up tables for the reallocation has the disadvantage that the number of combinations of tasks and processors is very large for even moderately sized systems [2] and grows exponentially after multiple processor failures. Although it is possible to use conventional algorithms to solve the problem, these methods are often computationally too expensive because of the hard real-time constraints and require an undesirable overhead because the algorithms have to be executed in a distributed environment without any hierarchical control.

Since finding the best allocation of tasks among the processors can be formulated as a constrained optimization problem, we will demonstrate how an optimization network can be used to solve this problem. The distributed system considered here resembles a simplified version of the SIFT computer and is based on a model described in [2], in which a conventional heuristic algorithm is used to solve this task allocation problem. We will later use this algorithm as a benchmark to assess the ANN-performance. The system has to execute n tasks and consists of m identical processors. Each task is replicated into r clones that are executed by different processors and submitted to a majority voter in order to detect and to mask possible hardware failures. Assuming periodic real-time tasks for a typical flight control system, the number of instructions per execution of task j, the frequency of execution and the execution rate of the processor determine the load that a certain task places on a processor, which is called the utilization z_j of task j. A particular allocation can be described by a variable V_{ij} with V_{ij}=1 if task j is scheduled on processor i and V_{ij}=0 otherwise. Then the variable $p_i = \sum_j z_j V_{ij}$ represents the overall load or utilization of processor i under the allocation V_{ij}.

The task allocation has to observe the constraint that each task must be executed by exactly r different processors in order to allow a majority vote. Additionally, the allocation should be done in a way that achieves at least an approximate load balancing among the processors. A load balancing in a distributed processing system is obviously desirable and Bannister and Trivedi [2] discuss several reasons why an imbalance potentially decreases the reliability of the system. It can be shown [2] that minimizing the sum of the squared processor utilizations $\sum p_i^2$ also minimizes the statistical variance of the p_i, which is a direct measure of the imbalance. We further assume that there are enough processors to accommodate a (balanced) assignment without capacity or scheduling violations.

The task allocation problem (TAP) is represented by an optimization network consisting of a two dimensional array of m×n neurons or elements, in which the output V_{ij} of an element is bounded between 0 and 1 and corresponds to the "hypothesis" that task j is

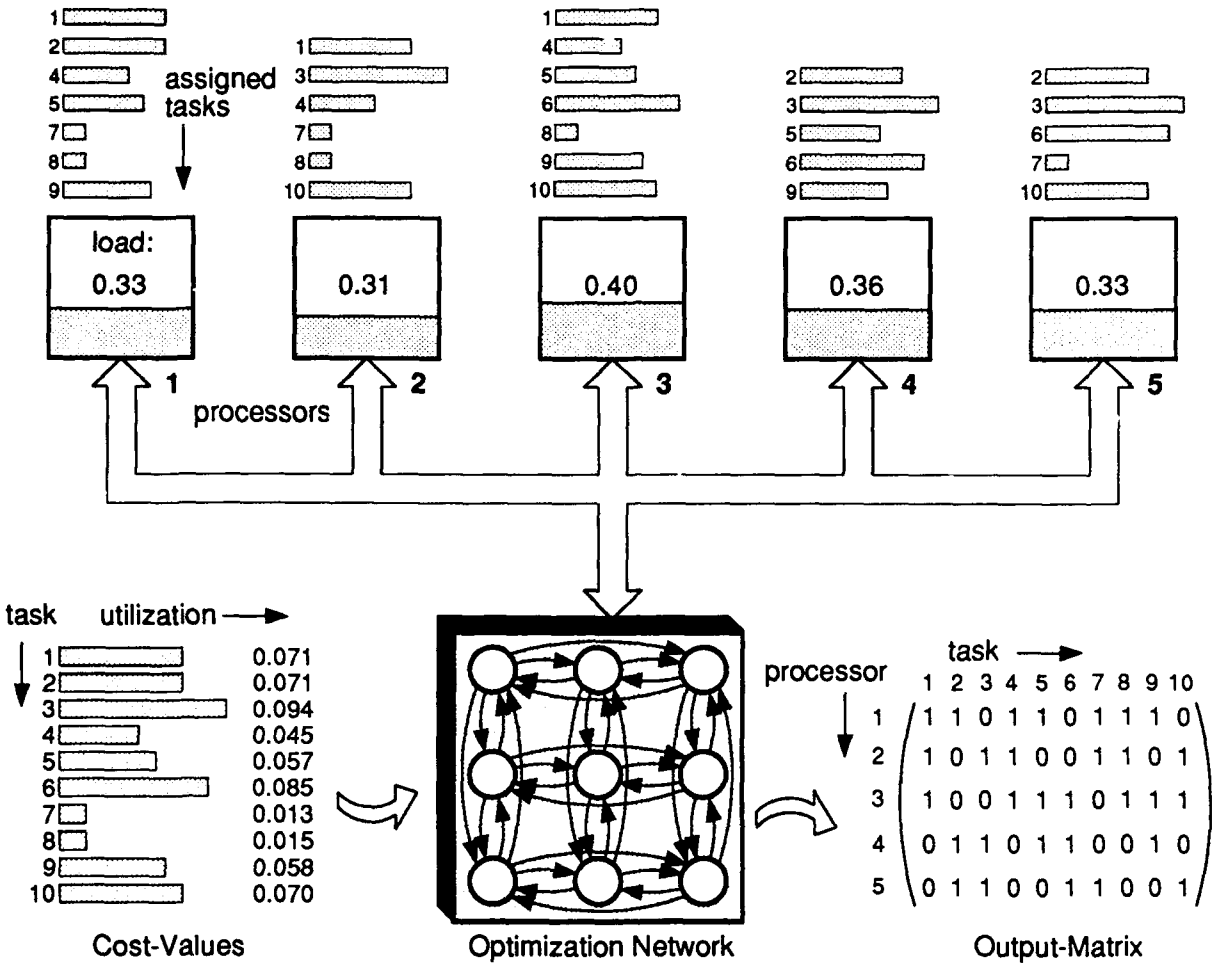

Figure 9. Example of an allocation of tasks to processors generated by an optimization network. Note that each task has to be executed by exactly three different processor while an approximate load balancing of the processors should be achieved.

assigned to processor i. Figure 9 illustrates this problem representation for an example in which 10 triplicated tasks are allocated to 5 processors. In order to map the task allocation problem onto the network, it has to be expressed as a function whose minima correspond to (local) solutions of the problem. With the above definitions, we can define the following energy function

$$E_{\mathrm{TAP}} = \frac{A}{2}\sum_{j=1}^{n}\left(\sum_{i=1}^{m}V_{ij}-r\right)^2+\frac{B}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}V_{ij}(1-V_{ij})+\frac{D}{2}\sum_{i=1}^{m}\left(\sum_{j=1}^{n}z_jV_{ij}\right)^2 \quad . \quad (16)$$

The first term in (16) has a minimum if the constraint is met (i.e. each task is executed by exactly r processors), the second term forces the outputs to converge to either 0 or 1, and the third term represents the cost-function to be minimized. Mapping (16) onto the energy

function (6) yields the following values values for the interconnections and the external current

$$T_{ij,lk} = -A\delta_{jk} + B\delta_{il}\delta_{jk} - Dz_j z_k \delta_{il}$$
$$I_{ij} = Ar - \frac{B}{2} \quad . \tag{17}$$

and the equations of motion

$$C_{ij}\frac{du_{ij}}{dt} = -\frac{u_{ij}}{R_{ij}} - A\sum_l V_{lj} + BV_{ij}$$
$$- Dz_j \sum_k z_k V_{ik} + Ar - \frac{B}{2} \quad . \tag{18}$$

We used the parameter values A=75, B=5, and D=350 as well as λ=25 and u_s=0 for the transfer function (1). Although there are only three parameters, we use D as the third parameter because we have previously associated D with the cost function of the problem. Our simulations are performed for different data-sets with task utilizations z_j randomly generated from a uniform distribution between 0.01 and 0.1. Because of the quadratic cost function in (16), the cost values z_j are part of the interconnections while the external current is constant. Thus, this problem is similar to the TSP and requires a random initialization to overcome the unstable equilibrium point at u_{ij}=0. We used the initial values V_{ij}=0.5+δ with small, uniform noise $-10^{-7} \le \delta \le 10^{-7}$. The equations of motions (18) were solved by Euler's method with a stepsize $\Delta t = 2\times10^{-5}$ and required an average of 5000 iterations to converge.

At this point, we can simulate the network and successfully "solve" the TAP as shown in Figure 9 with a performance that is comparable to the TSP-network, but this is not the actual task in this application. What is required is a *reallocation* of tasks *after a processor failure.* Therefore, the network has to be provided with the information of which processor has failed. Furthermore, it has to implement this information as an additional constraint before solving the problem. For example, the unavailability of a processor k can be represented by enforcing V_{kj}=0 for all j, that is, no tasks can be assigned to processor k. This additional constraint could be implemented either by external currents of sufficient strength to "shut down" all neurons in row k, or by switches connecting the outputs of all neurons in row k to "0" (ground potential). While the latter method seems to be somewhat crude, it has actually the advantage that a possible stuck-at-1 hardware fault of a neuron in that row is "overwritten" by the external switch. Producing this "short circuit" at the outputs is equivalent to our stuck-at-0 fault-injections in the last Section. There we have shown that the network indeed treats these "faults" as additional constraints to the optimization problem. Figure 10 illustrates the process of reallocation after a processor failure by using the same example shown in Figure 9.

The network is obviously a critical component of the system because a network failure would prevent the reconfiguration of the system after a processor failure, which leads to a

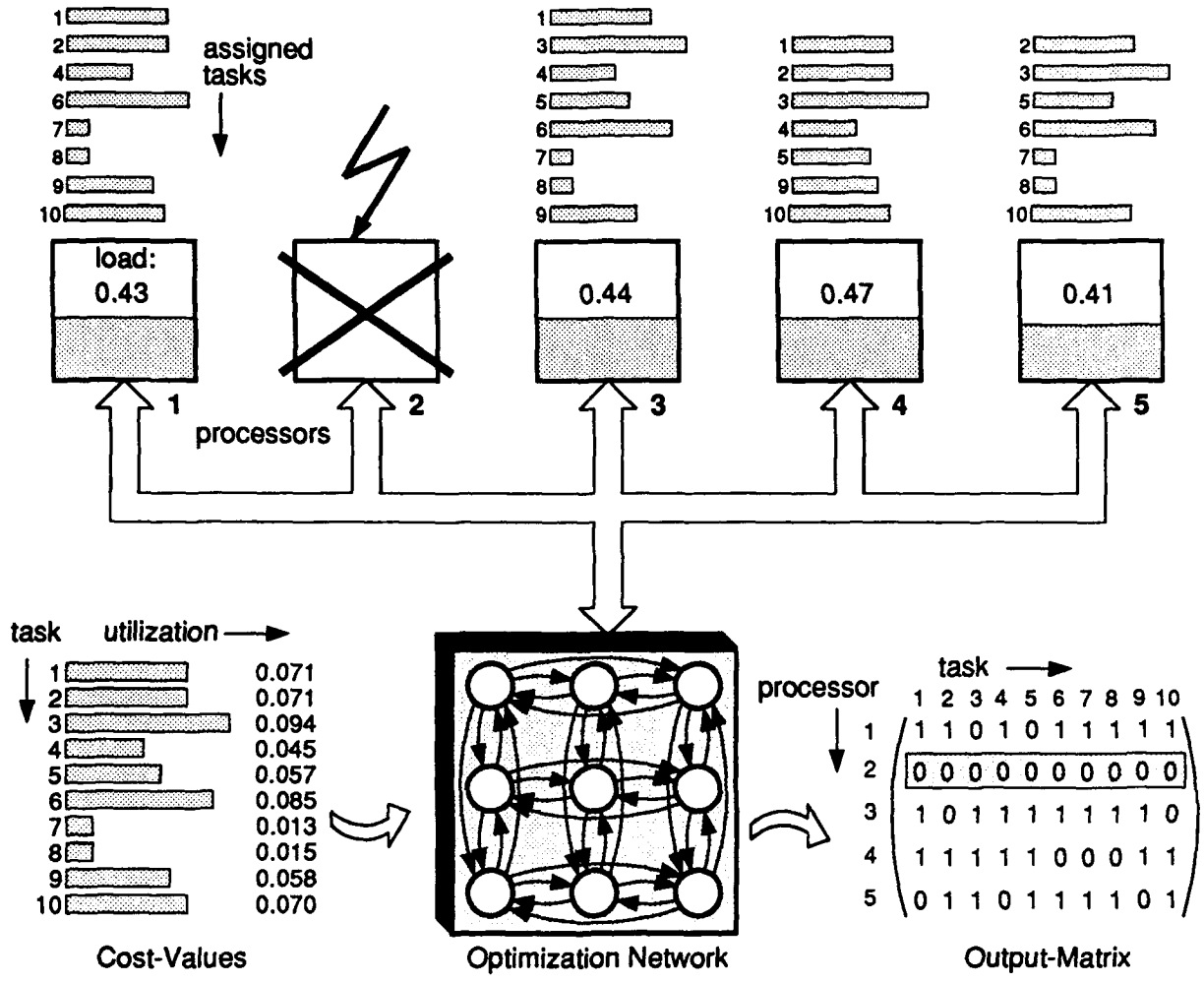

Figure 10. Example of a reallocation of tasks after a processor failure. The optimization network generates new allocations by observing the constraints and by approximately balancing the load of the processors.

total system failure. Thus, the fault-tolerance of the ANN becomes a crucial characteristic. We tested the fault-tolerance again by simulating stuck-at-0 and stuck-at-1 faults in randomly selected locations. Figure 11 illustrates the operation and the convergence of the network for the example of a system with m=7 processors and n=14 tasks where each task has to be executed by 3 different processors (r=3). Figure 11a shows the initialization of the (fault-free) network for a scenario in which processor 4 has failed, which is reflected by an output value of zero for all neurons in row 4. Figure 11b indicates the result after convergence with task 2, 3, and 6 assigned to processor 1, task 3, 5, and 7 assigned to processor 2, etc. The load balancing performance of the ANN is also illustrated in Figure 11b which lists the processor utilizations resulting from the ANN solution in comparison with a simple, heuristic reference algorithm [2]. As can be seen from the cost values listed at the bottom, which are the sum of the squares of the processor utilizations, the ANN is outperformed

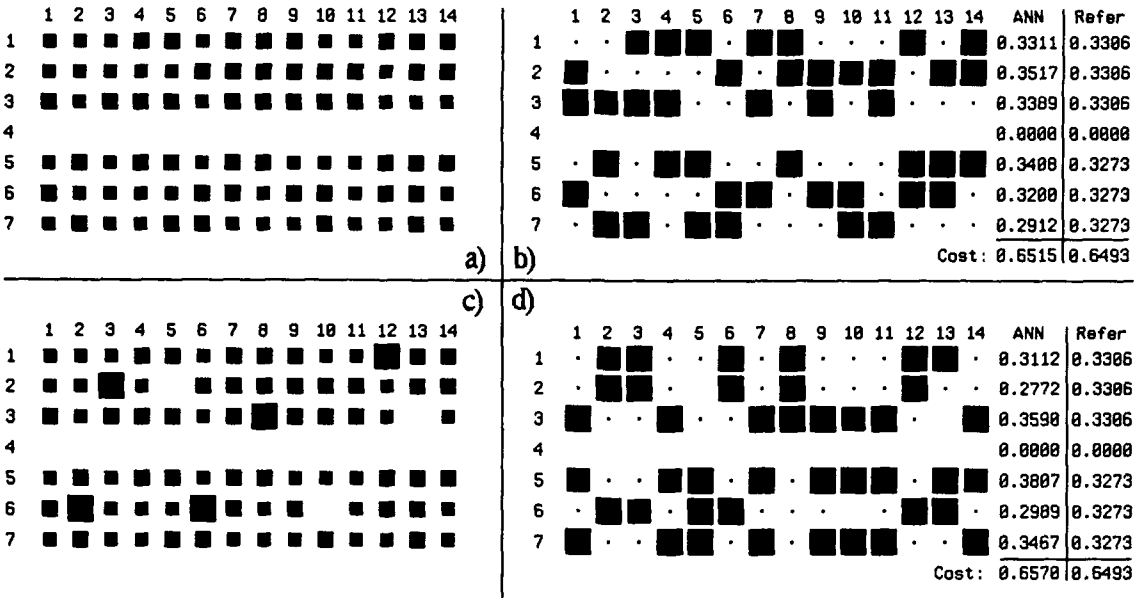

Figure 11. Illustration of the operation and convergence of a network generating a task allocation after the failure of processor 4 (m=7, n=14, r=3): a) initialization of the network (no faults), b) solution after convergence with resulting processor utilizations in comparison to the reference algorithm, c) initialization (five stuck-at-1 and three stuck-at-0 faults injected into the network), d) solution after convergence under the presence of the injected faults.

by the algorithm, although the difference of the values is only of the order of one percent. However, as we stated earlier, an approximate load balancing is sufficient in this case as long as the solution can be obtained fast and reliably.

The latter requirement is illustrated in Figure 11c. It shows the initialization of the network for the same scenario, but now with *eight* faults simultaneously present in the network. The fault locations of five stuck-at-1 and three stuck-at-0 faults are clearly visible after the initialization. Figure 11d shows the results after convergence and we can observe the same phenomenon that the faults do not impair the convergence, but act as additional constraints of the problem. According to the cost value in Figure 11d, the performance is only slightly worse than in the fault-free case.

Since the performance of the ANN varies considerably for different random initializations and different input data, it is necessary to evaluate the average performance over a sufficient number of problem instances in order to obtain a statistically relevant assessment. We simulated a system with m=8 processors and n=24 triplicated tasks (r=3), which requires a network of 192 neurons. Seven different test-sets of random task utilizations were generated. The network was simulated with seven different initializations for each test set. The solution quality q was used to assess the performance where values for c_{opt} were obtained from the heuristic algorithm in [2]. Figure 12 demonstrates the performance degradation

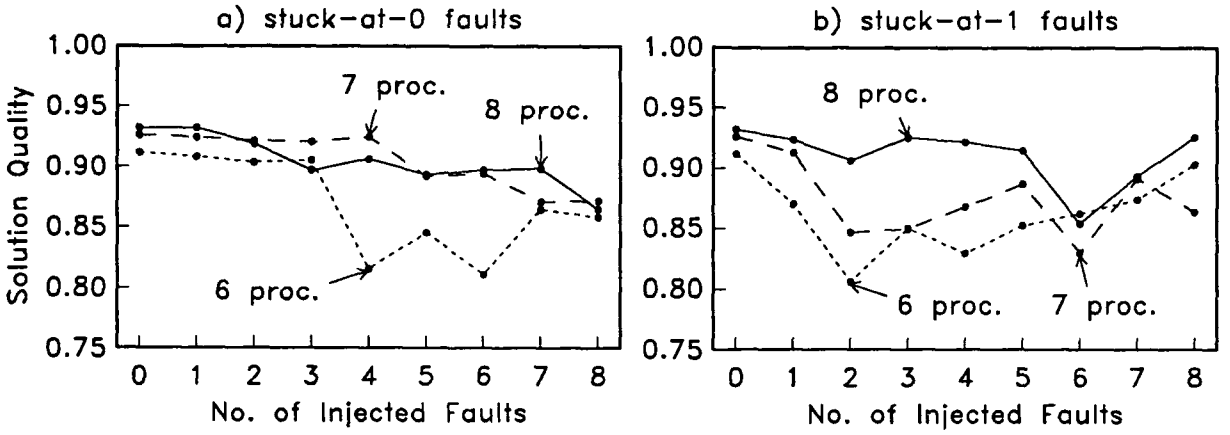

Figure 12. Performance degradation of the ANN allocating n=24 triplicated tasks (r=3) to m=8, 7, and 6 processors.

for up to 8 injected stuck-at-0 or stuck-at-1 faults, respectively. The number of processors refers to the remaining number of available processors in the system. For example, if the distributed system consists initially of 8 processors, then m=7 refers to the operation of the network after a failure of one processor with the neurons in the corresponding row switched to zero. Note that the solution quality in Figure 12 is plotted in the small range from 0.75 to 1.0, which magnifies the variations. As expected, the performance is very similar to the TSP because both use a quadratic cost function.

The results in Figure 12 confirm the qualitative observation in Figure 11 that the ANN exhibits an extreme fault-tolerance compared to conventional systems. Since the faults are randomly located and act as additional constraints of the problem, it is possible that one or more faults accidently "dictate" a better solution than the network would have found without faults. This explains the occasional *performance increase* after fault-injection and the nonmonotonic characteristic of the performance degradation. Of course, this is only possible because of the suboptimal performance of the ANN in the fault-free case. It is also important to note that none of the simulations converged to an invalid solution or to a solution that violates the capacity constraint $p_i<1$, although the latter was not explicitly enforced. An event that would lead to an invalid solution can only occur if there are more than r stuck-at-1 faults in the same column, thus assigning a task to more than r processors and violating the constraints. If the faults occur at random locations and if the failure rate of a stuck-at-1 fault is known for a particular hardware implementation, then this scenario can be used to estimate an upper bound for the reliability of the ANN.

7. Conclusion

The fault-tolerance of conventional systems is a carefully calculated design goal that requires some form of hardware- or software-redundancy, which increases the complexity

of the system. That is, it is always possible to build a simpler system without the redundancy, and this system has the same performance under fault-free conditions as the fault-tolerant system. In contrast, the fault-tolerance of optimization networks is inseparable from their functional characteristics and is neither planned nor can it be removed. We have demonstrated this "inherent" fault-tolerance in simulations and we have shown that the injected faults are treated by the network as additional constraints to the problem. While conventional systems often break down completely after a single fault, the network exhibits a graceful performance degradation even after multiple injected faults. This characteristic can be exploited and a fault-tolerant neural network integrated on a single analog VLSI chip might perform a critical task that would otherwise require a redundant microprocessor system with specially tested software.

As an example for a promising application, we used the neural network as a critical component of a fault-tolerant, distributed processing system. The failure of a processor requires a reconfiguration of the system and a reallocation of all tasks among the remaining processors. This task allocation has to observe certain constraints and should at least approximately balance the load of the processors. We showed how a neural network can solve this problem and demonstrated the robustness of the network by injecting simulated faults. Our results indicate that the network can indeed perform this task reliably and that even multiple faults do not impair the ability of the network to generate an answer with only slightly degraded performance.

In summary, we think that there exist applications for the type of neural network described in this paper that can take advantage of the speed, low weight, low power consumption, and fault-tolerance of future hardware implementations. However, in most cases, the actual performance of the network does not reach the performance of the best available, conventional optimization algorithm. Thus, the neural network approach is best suited to certain real-time applications that do not necessarily require the absolute best answer, but where it is necessary to generate an approximate answer fast and reliably. The characteristic of a graceful performance degradation without additional redundancy is especially interesting for applications such as long-term, unmanned space missions, where component failures have to be expected but no repair or maintenance can be provided.

References

[1] Anderson, J. A. Cognitive and psychological computation with neural models. *IEEE Transactions on Systems, Man, and Cybernetics SMC-13*, 5 (Sep-Oct 1983), 799–815.

[2] Bannister, J. A., and Trivedi, K. S. Task allocation in fault-tolerant distributed systems. In *Hard Real-Time Systems (Tutorial)*, J. A. Stankovic and K. Ramamritham, Eds. IEEE Computer Society Press, 1988, pp. 256–272.

[3] Belfore II, L. A., and Johnson, B. W. The fault-tolerance of neural networks. *The International Journal of Neural Networks - Research and Applications 1*, 1 (Jan 1989), 24–41.

[4] Brandt, R. D., Wang, Y., Laub, A. J., and Mitra, S. K. Alternative networks for solving the traveling salesman problem and the list-matching problem. In *Proceedings of the IEEE International Conference on Neural Networks, San Diego, CA* (July 1988), pp. II–333–340.

[5] Cohen, M. A., and Grossberg, S. Absolute stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE Transactions on Systems, Man, and Cybernetics SMC-13*, 5 (Sep/Oct 1983), 815–826.

[6] Garey, M. R., and Johnson, D. S. *Computers and Intractability*. W. H. Freeman, 1979.

[7] Grossberg, S. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural Networks 1*, 1 (1988), 17–61.

[8] Hedge, S., Sweet, J., and Levy, W. Determination of parameters in a Hopfield/Tank computational network. In *Proceedings of the IEEE International Conference on Neural Networks, San Diego, CA* (July 1988), pp. II–291–298.

[9] Hinton, G. E., and Sejnowski, T. J. Learning and relearning in Boltzmann machines. In *Parallel Distributed Processing, Vol. 1*, D. E. Rummelhart and J. L. McClelland, Eds. Bradford Books/MIT Press, 1986, ch. 7, pp. 282–317.

[10] Hinton, G. E., and Shallice, T. Lesioning a connectionist network: Investigations of acquired dyslexia. Technical Report CRG-TR-89-3, Dept. of Computer Science, University of Toronto, May 1989.

[11] Hopfield, J. J. Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. Natl. Acad. Sci. USA, Biophysics 81* (May 1984), 3088–3092.

[12] Hopfield, J. J., and Tank, D. W. "Neural" computation of decisions in optimization problems. *Biological Cybernetics 52* (1985), 141–152.

[13] Hutchinson, J. M., and Koch, C. Simple analog and hybrid networks for surface interpolation. In *Neural Networks for Computing*, J. S. Denker, Ed. American Institute of Physics, 1986, pp. 235–239.

[14] Lin, S., and Kernigham, B. W. An effective heuristic algorithm for the traveling salesman problem. *Operations Research 21* (1973), 498–516.

[15] Marcus, C. M., and Westervelt, R. M. Dynamics of analog neural networks with time delay. In *Advances in Neural Information Processing Systems*. Morgan Kauffman, 1989.

[16] Palumbo, D. L., and Butler, R. W. A performance evaluation of the software-implemented fault-tolerance computer. *J. Guidance 9*, 2 (March-April 1986), 175–180.

[17] Petsche, T., and Dickinson, B. W. Trellis codes, receptive fields, and fault tolerant, self-repairing neural networks. *IEEE Transactions on Neural Networks 1*, 2 (June 1990), 154–166.

[18] Protzel, P. W. Comparative performance measure for neural networks solving optimization problems. In *Proceedings of the International Joint Conference on Neural Networks IJCNN-90, Washington, D.C.* (January 1990), pp. II–523–526.

[19] Protzel, P. W., Palumbo, D. L., and Arras, M. K. Fault-tolerance of a neural network solving the traveling salesman problem. ICASE Report No. 89-12 / NASA Contractor Report 181798, ICASE / NASA Langley Research Center, Feb 1989.

[20] Sejnowski, T. J., and Rosenberg, C. R. NETtalk: a parallel network that learns to read aloud. Technical Report JHU/EECS-86/01, John Hopkins University, 1986.

[21] Smith, M. J., and Portmann, C. L. Practical design and analysis of a simple "neural" optimization circuit. *IEEE Transactions on Circuits and Systems 36*, 1 (January 1989), 42–50.

[22] Syslo, M. M., Deo, N., and Kowalik, J. S. *Discrete Optimization Algorithms*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1983.

[23] Tagliarini, G. A., and Page, E. W. A neural network solution to the concentrator assignment problem. In *IEEE Conference on "Neural Information Processing Systems – Natural and Synthetic", Denver, CO* (November 1987).

[24] Tank, D. W., and Hopfield, J. J. Simple "neural" optimization networks: An A/D converter, signal decision circuit, and a linear programming ciruit. *IEEE Transactions on Circuits and Systems CAS-33*, 5 (May 1986), 533–541.

[25] Van den Bout, D. E., and Miller, T. K. A traveling salesman objective function that works. In *Proceedings of the IEEE International Conference on Neural Networks, San Diego, CA* (July 1988), pp. II–299–304.

[26] Wilson, G. V., and Pawley, G. S. On the stability of the traveling salesman problem algorithm of Hopfield and Tank. *Biological Cybernetics 58* (1988), 63–70.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-187582 ICASE Report No. 91-45		

4. Title and Subtitle	5. Report Date
PERFORMANCE AND FAULT-TOLERANCE OF NEURAL NETWORKS FOR OPTIMIZATION	June 1991
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Peter W. Protzel Daniel L. Palumbo Michael K. Arras	91-45
	10. Work Unit No.
	505-90-52-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18605
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Michael F. Card

Submitted to IEEE Transactions on Neural Networks

Final Report

16. Abstract

One of the key benefits of future hardware implementations of certain Artificial Neural Networks (ANNs) is their apparently "built-in" fault-tolerance, which makes them potential candidates for critical tasks with high reliability requirements. This paper investigates the fault-tolerance characteristics of time-continuous, recurrent ANNs that can be used to solve optimization problems. The performance of these networks is first illustrated by using well-known model problems like the Traveling Salesman Problem and the Assignment Problem. The ANNs are then subjected to up to 13 simultaneous "stuck-at-1" or "stuck-at-0" faults for network sizes of up to 900 "neurons." The effect of these faults on the performance is demonstrated and the cause for the observed fault-tolerance is discussed. An application is presented in which a network performs a critical task for a real-time distributed processing system by generating new task allocations during the reconfiguration of the system. The performance degradation of the ANN under the presence of faults is investigated by large-scale simulations and the potential benefits of delegating a critical task to a fault-tolerant network are discussed.

17 Key Words (Suggested by Author(s))	18. Distribution Statement
Neural Networks, Fault-Tolerance, Optimization	62 - Computer Systems 63 - Cybernetics Unclassified - Unlimited

19 Security Classif. (of this report)	20 Security Classif. (of this page)	21 No. of pages	22 Price
Unclassified	Unclassified	33	A03

NASA FORM 1626 OCT 86